Object-Oriented Programming

Object-Oriented Programming

- **= Abstract Data Types**
  - package representation of data structure together with operations on the data structure
  - encapsulate internal implementation details
- **+ Inheritance**
  - support defining new ADT as incremental change to previous ADT(s)
  - share operations across multiple ADTs
- **+ Subclass Polymorphism**
  - allow variables to hold instances of different ADTs
- **+ Dynamic Binding**
  - run-time support for selecting right implementation of operation, depending on argument(s)

Some OO languages

- Simula 67: the original
- Smalltalk-80: popularized OO
- C++: OO for the hacking masses
- Java, C#: cleaned up, more portable variants of C++
- CLOS: powerful OO part of Common Lisp
- **Self**: very pure OO language
- **Cecil**, MultiJava, **EML**: OO languages from my research group
- Emerald, Kaleidoscope: other OO languages from UW

Abstract data types

- User-defined data structures along with user-defined operations
  - Support good specification of **interface** to ADT, hiding distracting implementation details
  - Prevent undesired dependencies between client and ADT, allowing implementation to change w/o affecting clients
  - Allow language to be extended with new types, raising & customizing the level of the language
- Called a **class** in OO languages
  - data structures called **objects**, or **instances** of the class
  - operations called **methods**; data called **instance variables**
- Modules have similar benefits

Inheritance

- Most recognizable aspect of OO languages & programs
- Define new class as *incremental modification* of existing class
  - new class is **subclass** of the original class (the **superclass**)
  - by default, **inherit** superclass's methods & instance vars
  - can add more methods & instance vars in subclass
  - can **override** (replace) methods in subclass
    - but not instance variables, usually

Example

```
class Rectangle {
  Point center;
  int height, width;
  int area() { return height * width; }
  void draw(OutputDevice out) { ... }
  void move(Point new_c) { center = new_c; }
  ...
}
class ColoredRectangle extends Rectangle {
  // center, height, & width inherited
  Color color;
  // area, move, etc. inherited
  void draw(OutputDevice out) { ... }  // override!
}
```

Benefits of inheritance

- Achieve more code sharing by **factoring** code into common superclass
  - superclass can be **abstract**
    - no direct instances, just reusable unit of implementation
  - encourages development of rich libraries of related data structures
- May model real world scenarios well
  - use classes to model different things
  - use inheritance for classification of things: subclass is a special case of superclass

7

Pitfalls of inheritance

- Inheritance often overused by novices
- Code gets fragmented into small factored pieces
- Simple extension & overriding may be too limited
  - e.g. exceptions in real-world classification hierarchies

8

Subclass polymorphism

- Allow instance of subclass to be used wherever instance of superclass expected
  - client code written for superclass also works/is reusable for all subclasses

```
void client(Rectangle r) {
    … r.draw(screen) …
}

ColoredRectangle cr = …;
… client(cr) …
// legal, because ColoredRectangle is a subclass of Rectangle
```

// but what version of draw is invoked?

9

Dynamic binding

- When invoke operations on object, invoke appropriate operation for *dynamic* class of object, not *static* class/type

```
ColoredRectangle cr = … ;
Rectangle r = cr;   // OK, because CR subclass of R
r.draw(); // invokes ColoredRectangle::draw!
```

- Also known as **message passing**, **virtual function calling**, **generic function application**

10

Method lookup

- Given a message `obj.msg(args)` …
- Start with run-time class `C` of `obj` (the **receiver**)
  - if `msg` is defined in `C`, then invoke it
  - otherwise, recursively search in superclass of `C`
  - if never find match, report run-time error ⇒ type checker guarantees this won't happen

11

Dynamic dispatching vs. static overloading

- Like overloading:
  - multiple methods with same name, in different classes
  - use class/type of argument to resolve to desired method
- Unlike overloading:
  - resolve using *run-time* class of argument, not *static* class/type
  - consider only receiver argument, in most OO languages
    - C++ & Java: regular static overloading on arguments, too
    - CLOS, Cecil, MultiJava: resolve using all arguments (**multiple dispatching**)

12

Example

- Without dynamic binding, use "typecase" idiom:

```
forall Shape s in scene.shapes do
  if s.is_rectangle() then rectangle(s).draw();
  else if s.is_square() then square(s).draw();
  else if s.is_circle() then circle(s).draw();
  else error("unexpected shape");
  end
end
```

- With dynamic binding, send message:

```
forall Shape s in scene.shapes do
  s.draw();
end
```

- What happens if a new `Shape` subclass is added?

13

Benefits of dynamic binding

- Allows subclass polymorphism and dynamic dispatching to class-specific methods
- Allows new subclasses to be added without modifying clients
- Allows more factoring of common code into superclass, since superclass code can be "parameterized" by "sends to **self**" that invoke subclass-specific operations
  - "Template method" design pattern

14

Pitfalls of dynamic binding

- Tracing flow of control of code is harder
  - control can pop up and down the class hierarchy
- Adds run-time overhead
  - space for run-time class info
  - time to do method lookup
    - but only an array lookup (or equivalent), not a search

15

Issues in object-oriented language design

- Object model:
  - hybrid vs. pure OO languages
  - class-based vs. classless (prototype-based) languages
  - single inheritance vs. multiple inheritance
- Dispatching model:
  - single dispatching vs. multiple dispatching
- Static type checking:
  - types vs. classes
  - subtyping
  - subtype-bounded polymorphism

16